ABLUR: An FPGA-based adaptive deblurring core for real-time applications

ABLUR: an FPGA-based adaptive deblurring core for real-time applications

Giuseppe Airò Farulla, Marco Indaco, Paolo Prinetto, Daniele Rolfo, Pascal Trotta
Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy
Email: {name.familyname}@polito.it
Telephone: (+39) 011.090-7191

***Abstract*—If a camera moves while taking a picture, motion blur is induced. There exist mechanical techniques to prevent this effect to occur, but they are cumbersome and expensive. Considering for example an Unmanned Aerial Vehicle (UAV) engaged in a save and rescue mission, where recording frames of scene to identify people and animals to rescue is required. In such cases, weight of equipments is of absolute importance, and no extra hardware can be used. In such case, vibrations are unavoidably transmitted to the camera, and recorded frames are affected by blur. It is then necessary to deblur in real-time every frame to allow post-processing algorithms to extract the largest possible amount of information from them. For more than 40 years, numerous researchers have developed theories and algorithms for this purpose, which work quite well but very often require multiple different versions of the input image, huge amount of computational resources, large execution times or intensive parameters tuning.**

We propose *ABLUR*, a novel self-adaptive core, implemented on a single Field Programmable Gate Array (FPGA) device, that is able to perform the deblurring task of single input images in real-time. The Dynamic Partial Reconfiguration (DPR) feature of modern FPGAs is exploited to enable self-adaptation of the deblurring algorithm parameters to the input images characteristics.

Experimental results show the limited amount of logic and memory resources required by the proposed hardware architecture.

I. Introduction

Nowadays, computer vision is one of the most evolving areas of Information Technology (IT). In every computer vision application, one or several images are taken from a camera, and processed, in order to extract information, used, for instance, for features identification [1], edge detection [2], or image registration [3].

However, there are cases in which it is not possible to rely on images' quality, as they may be affected by noise [4] or motion blur [5].

While capturing a frame, the camera must maintain the shutter opened for a finite amount of time, in order to acquire the proper amount of light and take a well defined image; relative movements between the camera and the scene during this interval induce motion blur in the captured image.

It is very difficult to obtain good results by processing blurry frames, and so input images must be firstly enhanced, in order to identify targets or extract information from them.

Restoring the *latent* image from the input blurry one has long been a challenging problem in digital imaging (e.g., [6], [7], [8]).

Authors have modelled the task as a two dimensional deconvolution process [9]. This simplification holds on when the blur is considered spatially invariant (or shift-invariant), meaning that every point in the original image spreads out the same way in forming the blurry image [10]. In this case, the blurry image is the result of the 2-D convolution of the real scene image with the *blur kernel*, also known as *Point Spread Function* (PSF) [11].

However, even in this simplistic case, to accomplish the deblurring task it is necessary to deal with 2-D deconvolution, that is well known to be an ill-conditioned and heavy task [12]. As 2-D convolution cannot be directly inverted, it is necessary to perform complex mathematical operations to retrieve the real image hidden behind the blurry input [13]. For this reason, deblurring algorithms are usually unable to achieve real-time performances.

Researchers in this field have always been more attracted by developing software than hardware accelerated solutions to the problem of deblurring, e.g., proposing interesting but slow solutions as in [14]. Very often, to obtain acceptable output results, a tuning phase is required in order to setup the deblurring algorithm parameters. In addition, a new setup phase is in general required when the input images characteristics change (e.g., due to contrast or brightness variations).

In literature, hardware is usually exploited as a medium to collect more in-depth information about the blurring procedure (e.g., by using sensors to detect the relative camera motion [15] [16]) rather than a way to speed-up mathematical calculations and, thus, software deblurring approaches.

Moreover, when dealing with real-time systems, a software implementation of complex algorithms cannot be used, since it does not meet the required performances. In this context, modern Field Programmable Gate Arrays (FPGAs) represent a good choice to hardware accelerate computational intensive software algorithms. FPGA-based implementations are often preferred to follow the current trend based on replacing Application Specific Integrated Circuits (ASICs) with more flexible FPGA devices, providing lower Non-Recurrent Engineering cost and time-to-market, even in mission-critical applications [17].

This paper proposes *ABLUR*, an adaptive hardware architecture that is able to deblur single input 1024*1024 frames in real-time, using a single reconfigurable FPGA device. The algorithm described in [18] has been choosen to be optimized and implemented on a modern FPGA to achieve high

performances, while obtaining high-quality outcomes. Main contribution of this paper is to exploit the Dynamic Partial Reconfiguration (DPR) feature (i.e, the ability to dynamically change selected portions of a circuit, while the rest of the design is left unchanged and fully functional [19] [20]) to adapt the algorithm parameters to the input images characteristics, that are not always predictable.
The paper is organized as follows: Section II presents an overview about existing deblurring approaches, Section III details the chosen deblurring algorithm and the proposed self-adaptive architecture, Section IV discusses the experimental results and, finally, Section V summarizes the contributions and obtained results.

II. Deblurring Algorithms Overview

The problem of developing an algorithm to recover latent images from blurry input ones has thrilled the scientific community during the last four decades; one of the very first works on this topic is presented in [21], where an iterative procedure is used for recovering a latent image that has been blurred by a known PSF.
Some of the most important contributions given to the state-of-the-art are here listed.
Classical deblurring approaches can be classified in blind and non-blind algorithms. While the former approach does not need any information about the blur kernel, the latter requires at least an estimated value. In any case, the problem is severely unconstrained [22].
Early works on deblurring usually model the blur kernel using simple shapes and priors, as in [23]. On the other hand, these exemplifications may lead to poor results when applied to natural images [18].
Linear motion blur kernel model used in many works is very often overly simplified for true motion blurring [24]. Authors in [25] state that the contents of real-world images can vary significantly across different frames or different patches in the same image. So, it is possible to learn various sets of bases from a pre-collected dataset of sample image patches and then, for a given patch to be processed, adapt one set of bases to characterize the local sparse domain.
During the last years, to consider more complex blurring models, several multi-image based approaches have been proposed. These methods estimate the blur kernel by analysing multiple images of the same scene [26], [27]. Although these approaches have the advantage of discarding too simplistic (and often unrealistic) assumptions, they cannot be applied when it is necessary to work on single input images. For example, [28] presents a hybrid camera system equipped with two imaging sensors. It can simultaneously captures high-resolution video together with a low-resolution one that has denser temporal sampling. Frames captured with higher temporal frequency are less affected by blur, since the smaller camera occlusion time is, the fewer relative movements between camera and scene are. Using the different information retrievable at the same moment from the two sensors, it is possible to deblur frames in the high resolution video and to contemporaneously estimate new high-resolution video frames from the low-resolution input ones.
Super-resolution (SR) is an imaging technique that leverages multiple low-resolution frames to construct a high-resolution frame [29]. It involves an exchange of information from frames basing on the assumption that the target has remained invariant. The majority of the work published on SR focuses on the mathematical algorithms behind SR and the ability to overcome inherent obstacles such as non-uniform blur [30], and motion estimation errors [31]. However, SR approaches are not suitable when a single standard camera is employed.
An interesting single-image deblurring approach based on Hyper-Laplacian priors is presented in [18]. Theoretical basis behind this method rely on the fact that typical gradients distributions in real-world scene images have been proven to be well modelled by a Hyper-Laplacian distribution ($p(x) \propto e^{k|x|^{\alpha}}$), with $0 < \alpha < 1$. However, the usage of such sparse distributions makes the problem more complex, thus slow to solve.
To speed-up the algorithm, authors in [18] present a method that splits the deblurring task into two separated sub-problems. Both these two phases aim at minimizing a cost function to retrieve the most probable latent image. This method proved to be very fast since the most time-consuming computations can be avoided by using a Look-Up-Table-based approach. However, it requires a heavy tuning phase before providing good quality outcomes.
For what concerns deblurring approaches, hardware acceleration has been mainly used for SR [32], [33]. To the best of our knowledge, we present for the first time a hardware implemented FPGA-based core, here called *ABLUR*, able to perform single-image deblurring in real-time. It exploits the algorithm presented in [18], avoiding human interaction during the algorithm parameters tuning phase, by self-adapting to the input images characteristics at run-time.

III. *ABLUR* Architecture

The aforementioned approach presented in [18] has been chosen because it has proven to be very fast and accurate; moreover, it is based on the Discrete Fourier Transform (DFT), an operation that is easily implementable in hardware [34]. We have developed a hardware architecture that is able to deblur single 1024x1024 pixels images in real-time (i.e., 25 frames-per-second, fps).
As explained in [18], the problem of restoring a latent image *x*, starting from the input blurry one *y*, can be solved in the frequency domain, exploiting Eq. 1.

$$x = \mathcal{F}^{-1}\left(\frac{\mathcal{F}(-f^1 \oplus w^1 - f^2 \oplus w^2) + \lambda \cdot \mathcal{F}(K)^* \cdot \mathcal{F}(y)}{\|\mathcal{F}(F^1)\| + \|\mathcal{F}(F^2)\| + \lambda \cdot \|\mathcal{F}(K)\|}\right), \quad (1)$$

where $\mathcal{F}(Z)$ and $\mathcal{F}^{-1}(Z)$ denote the two-dimensional direct and inverse DFT of a matrix Z, respectively [35], and $\|Z\|$ represents the matrix obtained by applying the modulus operator to each element of Z.
In Eq. 1, $*$ is the complex conjugate, $\oplus$ is the convolution operator and $\cdot$ denotes component-wise multiplication (the division is also performed component-wise), while λ is a weighting constant, choosen equal to 2000 in authors' MATLAB implementation[1]. Moreover, since this method belongs to the family of non-blind deblurring algorithms, it requires in input the blur kernel, represented with its Optical Transfer Function (OTF) K. The OTF models the transfer function of an optical system, and is represented as a matrix as big as the

[1] http://dilipkay.wordpress.com/fast-deconvolution/

input image [36].
Instead, F^1 and F^2 are the OTFs of f^1 and f^2, that are the two first-order derivative filters in the x and y axis, respectively ($f^1 = [1\ -1]$ and $f^2 = [1\ -1]^T$).
Finally, w^1 and w^2 are computed as:

$$\begin{aligned} w^1 &= \arg\min_w |w|^\alpha + \frac{1}{2}(w - v^1)^2 \\ w^2 &= \arg\min_w |w|^\alpha + \frac{1}{2}(w - v^2)^2, \end{aligned} \tag{2}$$

where

$$\begin{aligned} v^1 &= y \oplus f^1 \\ v^2 &= y \oplus f^2. \end{aligned} \tag{3}$$

In Eq. 2, α is a parameters related to the distribution of the gradients in the input image, and in general it is between $0 < \alpha < 1$ for real-world images, denoting a Hyper-Laplacian ditribution [18], while $\arg\min_z f(z)$ represents the values of z that minimize the function $f(z)$.
Authors propose to solve Eq. 2 by using a Look-Up Table (LUT), which, for a fixed α, stores pre-computed data (i.e., w^1 and w^2), for each possible v^i. Obviously, data are discretized, in order to limit the LUT size. In addition, they propose to compute off-line $\mathcal{F}(K)^*$ and the whole denominator from Eq. 1 as they do not depend on the input image y.
However, this algorithm presents two main limitations:

1) it is a non-blind deblurring algorithm, which implies that the exact blur kernel should be provided as an input parameter to correctly restore an image;
2) it requires a tuning phase that has major impacts on the final produced outcomes, as the value of α has to be fixed, for each input image.

To effectively implement this algorithm on an FPGA device, some considerations and optimizations have been done.
Concerning the first problem, from the knowledge of the system (e.g., vibrations induced to the camera or expected relative motion between camera and the scene), a generic estimation of the blur kernel can be employed as input. Tests have demonstrated that this algorithm is quite robust to errors in the initial kernel estimation, which can be fixed a-priori, and applied on each image at run-time (see Section IV).
To solve the second problem, we propose to estimate at run-time the distribution of the input image gradients, characterized by α, thus adapting the computations to the actual image scene characteristics.
It is worth noting that, with respect to Eq. (1), since the OTFs K, F^1 and F^2 are fixed a-priori, the denominator is fully off-line pre-computable thus, at run-time, it can be retrieved from an external memory.
Figure 1 shows the overall architecture of *ABLUR*.
ABLUR processes a stream of 8-bit packets representing a sequence of 1024x1024 grey scale frames, with 8 bit per pixel (bpp) resolution. It is assumed that the image pixels are received in a raster format, line-by-line from left to right and from top to bottom. *ABLUR* outputs a stream representing the deblurred input frames, with the same bpp resolution.
Several interfaces to external memories are also needed in order to store temporary data, that cannot be efficiently kept in the FPGA internal memory resources.

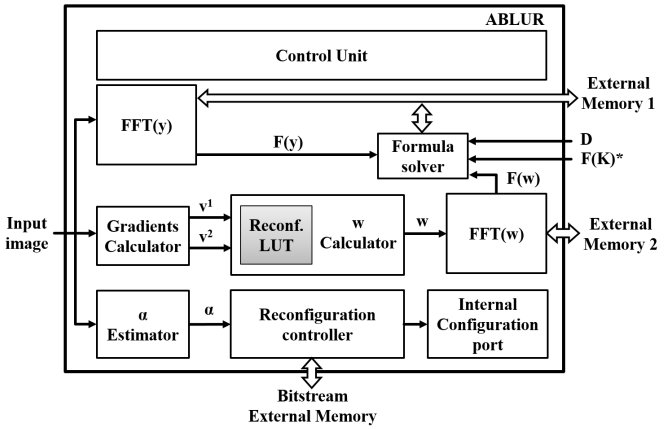


Figure 1. *ABLUR* block diagram

The following subsections detail all the main modules composing *ABLUR*.

A. Input Image Fast Fourier Transform module (FFT(y))

This module computes the two-dimensional Fast-Fourier Transform of the input image. Since the input image is 1024x1024 pixels, it outputs a matrix of 1024x1024 64-bit complex values (both the *real* and the *complex* parts of each value are represented on 32-bits).
In literature, many real-time FFT hardware modules have been presented (e.g., [37], [38]). Since the focus of this paper is not to present an architecture that implements the Fast Fourier Transform, this module has been implemented resorting to the Xilinx *LogiCore Fast Fourier Transform* core [39].
However, to compute a two-dimensional fourier transform, two phases must be performed. First, the FFT is computed for each row of the image, and stored in *External Memory 1*. Then, the final FFT results are computed by retrieving the temporary FFT data in a column order [40].
This module is also in charge of computing the Inverse FFT in Eq. 1, to extract the deblurred image results.

B. Gradient calculator

This module computes the gradients (i.e., v^1 and v^2) of the input image by convolving it with the filters f^1 and f^2 (see Eq. 3). Figure 2 shows the internal architecture of the *Grandients calculator* module.

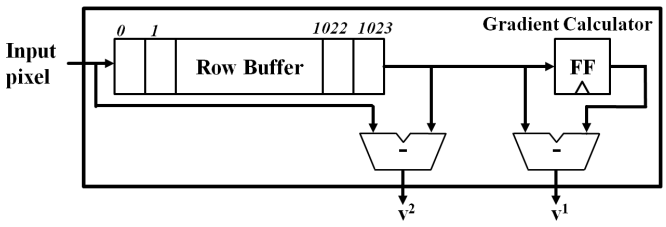


Figure 2. Gradient calculator architecture

For each pixel composing the input image, it outputs the associated gradients in the x and y axis (i.e., v^1 and v^2). Since the input images are received in a row-by-row raster

format, and the convolution with the filters f^1 and f^2 operates on adjacent pixels in the x and y axis, a First-In-First-Out (FIFO) buffer is needed to store a single 1024 pixels row of the input image (i.e., *Row Buffer* in Figure 2). This buffer has been implemented using a single FPGA internal Block-RAM (BRAM) memory resource [41]. At startup, the FIFO buffer is filled with all the pixels associated to the first row of the image. Then, whenever a new input pixels is received, it is stored inside the buffer. Leveraging the dual-port feature of the BRAMs, in the same clock cycle, the oldest stored pixel is read-out. The new read-out pixel is used, in conjuction with the last read-out one, stored in the register *FF* in Figure 2, to compute v^1. Simultaneously, the read-out pixel is subtracted to the actual received pixel to compute v^2.

C. α estimator

The α *estimator* module computes the α parameter (see Eq. 2) that best fits the characteristics of the input images. The resulting value of α is used to select the right configuration of the w *calculator* LUT, to be applied to the following image. This is acceptable since, at real-time frame rate (i.e., 25 fps), the image characteristics are very similar between the actual frame and the following one.

In particular, α characterizes the gradients distribution of the input image, that, for real-world image, follow a hyper-laplacian distribution (i.e., $p(x) \propto e^{|x|^\alpha}$ where $0 < \alpha < 1$) [18]. The distribution of the gradients can be computed by extracting the gradients histograms.

As shown in Figure 3, the α *estimator* is composed by two main sub-modules: (i) the *Histogram Calculator*, and (ii) the α *selector*.

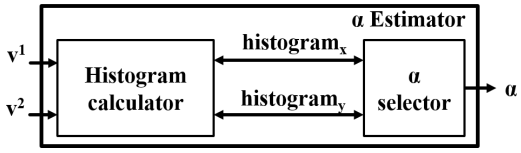


Figure 3. α estimator architecture

The *Histogram Calculator* computes the histogram of the input image gradients. Its internal architecture, shown in Figure 4, is based on two dual-port BRAM buffers (i.e., $BRAM_x$ and $BRAM_y$), each one associated to a 20-bit counters.

The values of v^1 and v^2, received from the *Gradients calculator*, are used to address the two buffers. Within the same clock cycle, the two read-out values are incremented by one, and stored in the same address location of the respective buffer. During this phase the α *read* signal is set to 0 by the *Controller*. When all v^1 and v^2 values are received, the *Controller* sets the *HD* signal, indicating that the two buffers contain the complete histograms associated to the gradients in the x and y directions.

The α *selector*, using a Look-Up Table (LUT) approach, outputs the α value that best fits the computed histogram distribution. In particular, it contains the α*LUT*, as shown in Figure 5, which stores 12 α values in the range (0.40, 0.95), discretized with a step of 0.05. Figure 6 plots the hyper-laplacian distributions associated to some α stored in the α*LUT* and their average slopes in the ranges $[-30, -20]$ and $[20, 30]$.

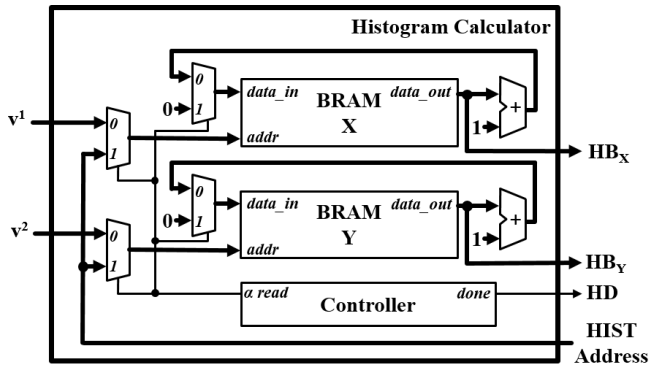


Figure 4. Histogram calculator architecture

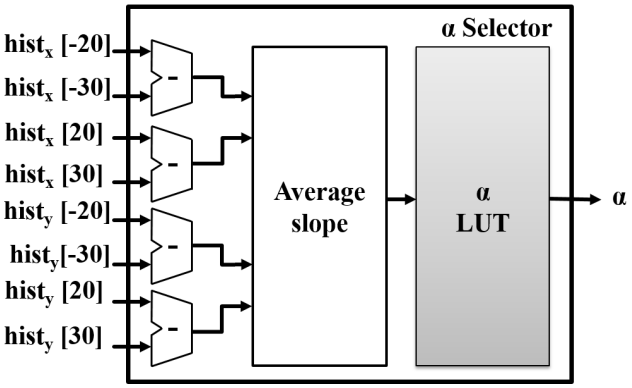


Figure 5. α selector internal architecture

As can be noted from Figure 6, looking to the slopes of the functions in the ranges $[-30, -20]$, or $[20, 30]$, is sufficient to discriminate between hyper-laplacian functions with different α values. Thus, the α *selector* reads from both histogram buffers the values of the histogram bars, associated to the gradient values 20, -20, 30, and -30, only. To accomplish this task, the *Controller* of the *Histogram Calculator* sets α *read* to 1, while the *HIST Address* signal is used by the α *selector* to extract the histogram bar values HB_x and HB_y, associated to the aforementioned gradient values.

Then, the average slope of the hyper-laplacian function in the selected range is computed and used to address the α*LUT* in order to extract the α parameter (Figure 5).

It is worth noting that, although only few values are used, we have chosen to compute the whole gradients histograms since these information are often exploited by subsequent image processing algorithms (e.g, for edge detection [42]), thus they can be an additional output of *ABLUR*. In any case, this computation does not affect the overall performances, and it requires very few resources.

D. Reconfiguration Manager

The *Reconfiguration Manager* receives in input the α, computed by the α *estimator*, and retrieves, from an external memory the partial configuration bitstream associated to the new chosen configuration for the LUT in the w *calculator*. The

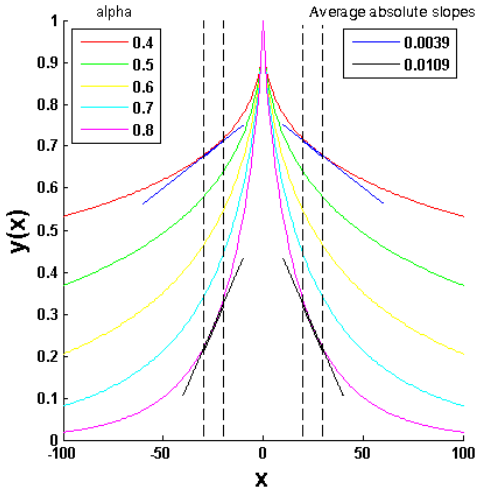


Figure 6. Hyper-Laplacian distributions with different α values

configuration bitstream is then written to the internal configuration port (i.e., *ICAP* in Xilinx FPGAs [43], or Configuration Control Block in Altera ones [20]), located inside the FPGA device, and directly connected to its configuration memory.
At the end of the reconfiguration process the w *calculator* LUT contains the updated values, corresponding to the selected α, that can be used to compute w during the next image cycle.

E. w calculator

The w *calculator* module operates in two consecutive steps. First, it solves Eq. 2 using a LUT approach. Basically, it receives v^1 and v^2; as the LUT stores the corresponding values of w for discretized v values, it is possible to compute w^1 and w^2 very fast. For each different value of α a different LUT is required (as Eq. 2 depends on α).
To ensure a good approximation, for a fixed α, the LUT contains 10^4 w^1 and w^2 32-bits values, as proposed in [18], leading to the usage of two 312,5 Kbits memories (each one used to compute w^1 and w^2, respectively).
In order to save FPGA internal memory resources, at run-time, only the LUT associated with the actual estimated value of α is instantiated inside the FPGA device. Run-time partial reconfiguration is then exploited to change the LUT configuration when the α value changes.
Afterwards, w^1 and w^2 are convolved with the negated values of the filters f^1 and f^2, using the same architecture as in Fig 2. Finally, the two convolved values are added togheter (see Eq. 1) to calculate the value of the w that is the output of this module.

F. w Fast Fourier Transform module (FFT(w))

This module computes the two-dimensional Fast-Fourier Transform of the values received from the w *calculator*. It is important to note that the received values represent a 1024x1024 elements matrix.
As the *FFT(y)* module, the *FFT(w)* has been implemented resorting to the Xilinx *LogiCore Fast Fourier Transform* core [39].

G. Formula Solver

The *Formula Solver* module is in charge of computing the sums and the component-wise division required by Eq. 1. This module receives $\mathcal{F}(y)$ and $\mathcal{F}(w)$ as inputs, and reads an external memory to retrieve both $\mathcal{F}(K)^*$ and the whole denominator D, which are pre-computed off-line (see Section III).
This module outputs a 1024x1024 complex matrix on which will be applied the inverse Fourier Trasform to retrieve the deblurred output image.

H. Control Unit

This module coordinates the operations of all the aforementioned modules. In fact, *ABLUR* operations can be grouped in four phases.
During the first phase, while the input image is received, the *FFT(y)*, the *Gradients Calculator*, the w *calculator*, the *FFT(w)*, and the α *estimator* modules are activated. In particular, *FFT(y)* and *FFT(w)* compute the first part of the two-dimensional Fourier Transform, on the rows of the respective input matrices (as mentioned in Section III-A and Section III-F), while α *estimator* computes the gradients histograms.
In the second phase, when the image is completely received, *FFT(y)* and *FFT(w)* computes the second part of the Fourier Transforms, retrieving the data computed during the first phase. In the meanwhile, α *estimator* outputs the α value. During this phase, the *Formula Solver* receives in input all the data needed to compute the sums and the division in Eq. 1.
In the third phase the *FFT(y)* module is used to compute the first part of the inverse Fourier Transform of the values extracted by the formula solver, while the *Reconfiguration Controller* reconfigures the w *calculator LUT* with the chosen configuration, reading the estimated α value.
Finally, in the fourth phase, the same module computes the second part of the inverse Fourier Transform and outputs the deblurred image values.

IV. Experimental results

To evaluate the hardware resources usage and the timing performances of the proposed architecture, *ABLUR* has been synthesized and implemented, resorting to *Xilinx ISE Design Suite 14.6*, on a Xilinx *Virtex 7 VX485T* FPGA device. Post place-and-route simulations have been done using *Modelsim SE 10.0c* to annotate the switching activities of internal nodes, and *Xilinx XPower Analyzer* has been exploited to estimate the overall power consumption.
Table I reports the FPGA resources usage of each internal module, along with the percentages of consumed resources with respect to the ones available in the selected device.
From Table I it is possible to note the limited hardware resources consumption, in terms of both logic (i.e., LUTs and *Digital Signal Processors* (DSPs)) and memory resources (i.e. BRAMs), for the selected device.
The 37 internal memory resources consumed by the w *calculator* are needed to store the *Reconfigurable LUT* associated to the run-time selected α value. The reconfiguration of this Look-Up Table requires 0.2 ms, since the configuration bitstream is about 80 KBytes, and the maximum bandwidth of the internal reconfiguration port (called *ICAP* in Xilinx devices) is equal to 3.2 Gbit/s [43]. This reconfiguration time does

Table I. Resource Usage for *Xilinx Virtex 7 VX485T FPGA device*

Module	FPGA Area Occupation			
	LUTs	FFs	BRAMs	DSPs
Control Unit	1,347	113	-	-
FFT(y)	2,207	376	16	94
FFT(w)	2,207	376	16	94
Gradients Calculator	112	35	1	-
α estimator	315	34	2	-
w calculator	265	53	37	-
Reconfiguration controller	150	66	-	-
Formula Solver	2,113	560	-	4
Total	**8,716 (2.87%)**	**1,613 (0.27%)**	**72 (3.50%)**	**192 (6.86%)**

not influence the overall throughput, since the reconfiguration process can be performed while carrying out the final inverse Fourier transform, that is more time consuming.

At the maximum operating frequency of 255 MHz, *ABLUR* is able to process 29 1024x1024 frames per second, thus achieving real-time performances. This working frequency leads to a power consumption of about 664 mW, where the major contribution is given by the clock generation and distribution circuitry (1 *Digital Clock Manager* FPGA internal resource is used to generate the system clocks [44]) and by the internal *Digital Signal Processors* (DSPs) modules [45].

To demonstrate the effectiveness and to quantify the accuracy of the proposed self-adapting approach, a test environment has been developed to read sharp natural-world images and injecting motion blur. The proposed architecture has been modeled as a Matlab script resorting to a fixed-point algebra to emulate the actual hardware precision. The Matlab model has been used also to perform functional verification of the implemented hardware architecture.

During the test phase, a motion blur kernel was used to simulate relative movements between camera and scene that are 7-pixel long and with an angle of 3 degrees with the x axis. The test environment is based on Matlab R2012b, running on Windows 7 x64 on a Notebook PC equipped with an Intel Core i5-2450M @2.50GHz CPU and 8 GB of RAM.

After injecting blur in the original images, the test environment invokes the deblurring function, implementing in software the algorithm executed by *ABLUR*. The blur kernel k passed to the algorithm is a minor perturbation of the true kernel, to mimic kernel estimation errors, as done in [18].

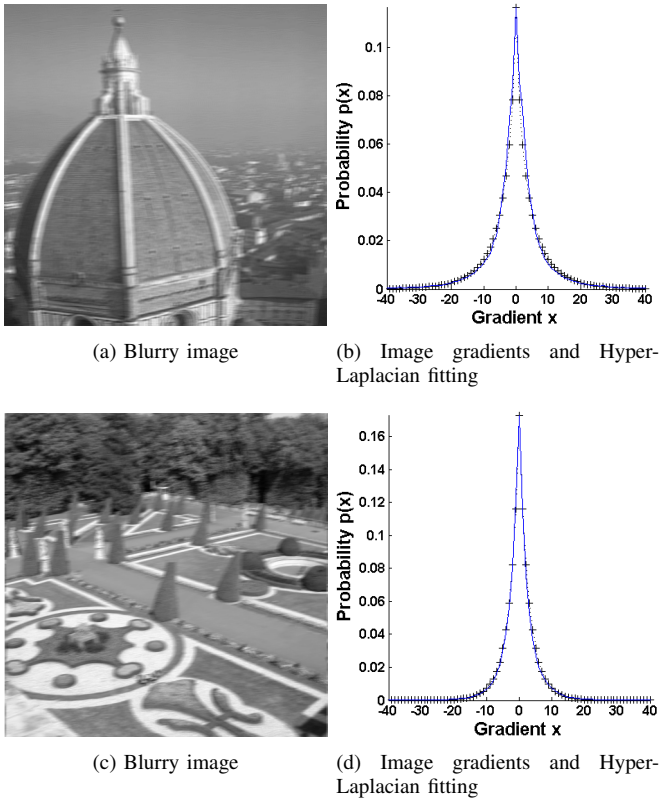
Tests have been performed on 100 1024x1024 pixels images. For each image, a software routine finds the α value that best fits the image gradients distribution. This value is also the one that minimizes the error between the reconstructed latent image, and the original input one. In particular, to quantify the quality of the reconstructed images, we used the Root Mean Square Error (RMSE), computed as:

$$\mathrm{RMSE}(L,O) = \sqrt{\frac{\sum_{i=1}^{1024}\sum_{j=1}^{1024}(L_{i,j}-O_{i,j})^2}{1024\cdot 1024}} \qquad (4)$$

where $L_{i,j}$ and $O_{i,j}$ represent a pixel in position (i,j) in the latent and original images, respectively.

Figure 7 shows two images examples along with their hyper-laplacian gradients distributions, characterized by two different α values.

Instead, the graph in Figure 8 shows the RMSE results while applying the algorithm implemented in *ABLUR*, with different



(a) Blurry image (b) Image gradients and Hyper-Laplacian fitting

(c) Blurry image (d) Image gradients and Hyper-Laplacian fitting

Figure 7. Real-world scene images affected by blur and their gradients distribution, together with the Hyper-Laplacian that better fits them (represented with black crosses)

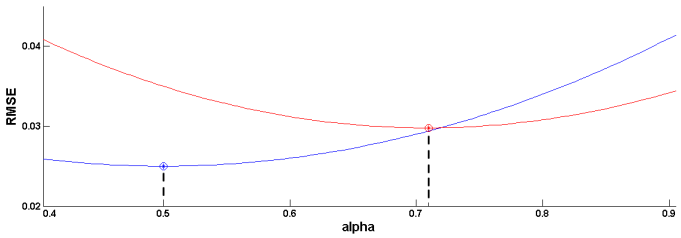


Figure 8. RMSE of the recovered latent images w.r.t. the original ones, varying the input α value, for the two examples in Figure 7 (the minimum RMSE is highlighted with a circled star)

α values.

It can be noted that the optimal α value is different between the two images, and correspond to the ones that characterize their Hyper-Laplacian gradients distribution.

During simulations *ABLUR* was able to identify the optimal α value, with a 0.05 resolution, thus ensuring equals, or even better outcomes w.r.t using a static α input.

In addition, since the hardware implementation of *ABLUR* uses fixed-point data representation, we evaluated the error introduced w.r.t. using a software implemented double precision version of the same algorithm. Figure 9 shows the visual results and the RMSE values of *ABLUR* and software double precision version outputs.

For the sake of completeness, the output results of *ABLUR* have been compared with the ones obtained by other single-

(a) Latent image restored by *ABLUR*

(b) Latent image restored by double precision software algorithm

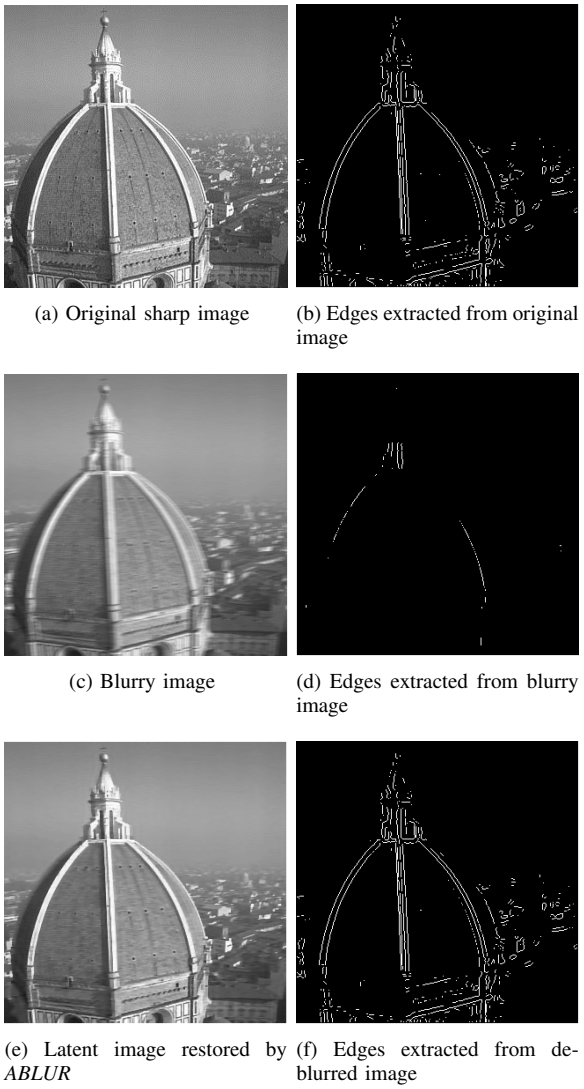
Figure 9. Example from Figure 7 deblurred by *ABLUR* (RMSE=0.044) and by software implemented double precision version of the same algorithm (RMSE=0.039)

image deblurring approaches (i.e., [18] and two MATLAB built-in functions *Deconvlucy* and *Deconvblind*, both based on the algorithm discussed in [21]). Results are summarized in Table II, and show that ABLUR achieves real-time performances while still providing high quality outcomes. Slight worsening in RMSE are due to approximations of the considered fixed-point algebra. The average elapsed time and the average RMSE are computed over 100 runs.

Table II. Comparison among deblurring approaches in terms of execution time and RMSE

Algorithm	Avg Elapsed Time (s)	AVG RMSE
ABLUR (HW)	0,034	0,0574
[18]	2,094	0,0409
Deconvlucy	3,126	0,0454
Deconvblind	6,396	0,0455

ABLUR ensures a speed-up of about 60x with respect to the Matlab version of the algorithm proposed in [18], while providing still acceptable results.
Deconvlucy and *Deconvblind* provide similar results in terms of RMSE, while being more time consuming w.r.t. the approach exploited by *ABLUR* [18].
Finally, we briefly discuss a possible example of usage of *ABLUR*. Consider an Unmanned Aerial Vehicle (UAV) engaged in a save and rescue mission, recording frames of scene to identify people to rescue while flying. To automatically detect people in difficulties, it could be useful to detect edges in every frames; such edges may be compared to typical human shapes, so that an alarm is triggered when possible human target is found. However, in such case, vibrations are unavoidably transmitted to the camera, and recorded frames are affected by blur, so that small edges are confused (or even totally hidden) by blur and impossible to detect. It is then necessary to deblur in real-time every frame to allow post-processing algorithms to extract the largest possible amount of sharp edges from them. Figure 10 shows the outcome of an edge-detection algorithm applied on the original image, its blurry version and the the latent image recovered by *ABLUR*. As is highlighted in this example, edges are definitely more sharp and detailed when extracted from the deblurred image, and very similar to the ones extracted from the original image.



(a) Original sharp image

(b) Edges extracted from original image

(c) Blurry image

(d) Edges extracted from blurry image

(e) Latent image restored by *ABLUR*

(f) Edges extracted from deblurred image

Figure 10. Example from Figure 7 deblurred by *ABLUR* and edges extracted from blurry and deblurred image

V. Conclusion

This paper presented *ABLUR*, a high performance FPGA-based adaptive deblurring core for real-time applications. *ABLUR* is able to self-adapt the deblurring parameters to the characteristics of the input image, resulting in more accurate outcomes.
Experimental results show the limited FPGA hardware resources consumption and an improvement of the quality of the recovered latent image w.r.t. the one obtained from a static deblurring approach. These enhancements allow better precision of all the following image processing modules (e.g., edge detector), that receive in input the deblurred image.

References

[1] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proc. of the 4th Alvey Vision Conference*, 1988, pp. 147 – 151.

[2] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679 – 698, 1986.

[3] G. Troglio, J. Le Moigne, J. Benediktsson, G. Moser, and S. Serpico, "Automatic extraction of ellipsoidal features for planetary image registration," *IEEE Geoscience and Remote Sensing Letters*, vol. 9, no. 1, pp. 95 –99, 2012.

[4] S. Di Carlo, P. Prinetto, D. Rolfo, and P. Trotta, "AIDI: An adaptive image denoising fpga-based ip-core for real-time applications," in *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, June 2013, pp. 99–106.

[5] L. O. Russo, G. Airò Farulla, M. Indaco, S. Rosa, D. Rolfo, and B. Bona, "Blurring prediction in monocular slam," in *International Design & Test Symposium. 2013. Proceedings. 2013 International IEEE Conference on*.

[6] O. Whyte, J. Sivic, A. Zisserman, and J. Ponce, "Non-uniform deblurring for shaken images," *International Journal of Computer Vision*, vol. 98, no. 2, pp. 168–186, 2012.

[7] W. Wang and M. K. Ng, "On algorithms for automatic deblurring from a single image," *Journal of Computational Mathematics*, vol. 30, no. 1, pp. 80–100, 2012.

[8] X. Chen, X. He, J. Yang, and Q. Wu, "An effective document image deblurring algorithm," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE, 2011, pp. 369–376.

[9] J.-F. Cai, H. Ji, C. Liu, and Z. Shen, "Framelet-based blind motion deblurring from a single image," *Image Processing, IEEE Transactions on*, vol. 21, no. 2, pp. 562–572, 2012.

[10] M. M. Sondhi, "Image restoration: The removal of spatially invariant degradations," *Proceedings of the IEEE*, vol. 60, no. 7, pp. 842–853, 1972.

[11] M. Cannon, "Blind deconvolution of spatially invariant image blurs with phase," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 24, no. 1, pp. 58–63, 1976.

[12] P. Campisi and K. Egiazarian, *Blind image deconvolution: theory and applications*. CRC press, 2007.

[13] D. Krishnan, T. Tay, and R. Fergus, "Blind deconvolution using a normalized sparsity measure," in *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE, 2011, pp. 233–240.

[14] Z. Hu, J.-B. Huang, and M.-H. Yang, "Single image deblurring with adaptive dictionary learning," in *Image Processing (ICIP), 2010 17th IEEE International Conference on*. IEEE, 2010, pp. 1169–1172.

[15] Y.-W. Tai, P. Tan, and M. S. Brown, "Richardson-lucy deblurring for scenes under a projective motion path," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 33, no. 8, pp. 1603–1618, 2011.

[16] N. Joshi, S. B. Kang, C. L. Zitnick, and R. Szeliski, "Image deblurring using inertial measurement sensors," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 30, 2010.

[17] S. Habinc, "Suitability of reprogrammable FPGAs in space applications - feasibility report," Gaisler Research, Tech. Rep., 2002.

[18] D. Krishnan and R. Fergus, "Fast image deconvolution using hyper-laplacian priors," in *Advances in Neural Information Processing Systems*, 2009, pp. 1033–1041.

[19] Xilinx Corporation, *Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite - WP374*, Internet, 2012.

[20] Altera Corporation, *Design Planning for Partial Reconfiguration*, Internet, 2013.

[21] L. Lucy, "An iterative technique for the rectification of observed distributions," *The astronomical journal*, vol. 79, p. 745, 1974.

[22] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman, "Understanding and evaluating blind deconvolution algorithms," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 1964–1971.

[23] G. Pavlovic and A. M. Tekalp, "Maximum likelihood parametric blur identification based on a continuous spatial domain model," *Image Processing, IEEE Transactions on*, vol. 1, no. 4, pp. 496–504, 1992.

[24] J.-F. Cai, H. Ji, C. Liu, and Z. Shen, "Blind motion deblurring from a single image using sparse approximation," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009, pp. 104–111.

[25] W. Dong, L. Zhang, G. Shi, and X. Wu, "Image deblurring and super-resolution by adaptive sparse domain selection and adaptive regularization," *Image Processing, IEEE Transactions on*, vol. 20, no. 7, pp. 1838–1857, 2011.

[26] J. Chen, L. Yuan, C. Tang, and L. Quan, "Robust dual motion deblurring," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE, 2008, pp. 1–8.

[27] A. Rav-Acha and S. Peleg, "Two motion-blurred images are better than one," *Pattern Recognition Letters*, vol. 26, no. 3, pp. 311–317, 2005.

[28] Y.-W. Tai, H. Du, M. S. Brown, and S. Lin, "Correction of spatially varying image and video motion blur using a hybrid camera," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, no. 6, pp. 1012–1028, 2010.

[29] B. Leung and S. O. Memik, "Exploring super-resolution implementations across multiple platforms," *EURASIP Journal on Advances in Signal Processing*, vol. 2013, no. 1, p. 116, 2013.

[30] M. Elad and Y. Hel-Or, "A fast super-resolution reconstruction algorithm for pure translational motion and common space-invariant blur," *Image Processing, IEEE Transactions on*, vol. 10, no. 8, pp. 1187–1193, 2001.

[31] S. Farsiu, M. D. Robinson, M. Elad, and P. Milanfar, "Fast and robust multiframe super resolution," *Image processing, IEEE Transactions on*, vol. 13, no. 10, pp. 1327–1344, 2004.

[32] M. E. Angelopoulou, C.-S. Bouganis, P. Y. Cheung, and G. A. Constantinides, "Fpga-based real-time super-resolution on an adaptive image sensor," in *Reconfigurable Computing: Architectures, Tools and Applications*. Springer, 2008, pp. 125–136.

[33] T. Szydzik, G. M. Callico, and A. Nunez, "Efficient fpga implementation of a high-quality super-resolution algorithm with real-time performance," *Consumer Electronics, IEEE Transactions on*, vol. 57, no. 2, pp. 664–672, 2011.

[34] I. S. Uzun, A. Amira, and A. Bouridane, "Fpga implementations of fast fourier transforms for real-time signal and image processing," in *Vision, Image and Signal Processing, IEE Proceedings-*, vol. 152, no. 3. IET, 2005, pp. 283–296.

[35] R. N. Bracewell and R. Bracewell, *The Fourier transform and its applications*. McGraw-Hill New York, 1986, vol. 31999.

[36] J. W. Goodman, *Introduction to Fourier optics*. Roberts and Company Publishers, 2005.

[37] I. S. Uzun, A. Amira, and A. Bouridane, "Fpga implementations of fast fourier transforms for real-time signal and image processing," in *Vision, Image and Signal Processing, IEE Proceedings-*, vol. 152, no. 3. IET, 2005, pp. 283–296.

[38] A. E. Desjardins, B. J. Vakoc, M. J. Suter, S.-H. Yun, G. J. Tearney, and B. E. Bouma, "Real-time fpga processing for high-speed optical frequency domain imaging," *Medical Imaging, IEEE Transactions on*, vol. 28, no. 9, pp. 1468–1472, 2009.

[39] Xilinx Corporation, *LogiCORE IP Fast Fourier Transform v9.0 Product Guide - PG109*, Internet, 2013.

[40] I. Pitas, *Digital image processing algorithms and applications*. Wiley.com, 2000.

[41] Xilinx Corporation, *7 Series FPGAs Memory Resources User Guide - UG473*, Internet, 2014.

[42] D. Sen and S. K. Pal, "Gradient histogram: Thresholding in a region of interest for edge detection," *Image and Vision Computing*, vol. 28, no. 4, pp. 677–695, 2010.

[43] Xilinx Corporation, *Partial Reconfiguration User Guide - UG702*, Internet, 2013.

[44] Xilinx Corporation, *7 Series FPGAs Clocking Resources User Guide - UG472*, Internet, 2013.

[45] Xilinx Corporation, *7 Series FPGAs DSP48E1 Slice User Guide - UG479*, Internet, 2013.